

Code Generation

By Birku L.

Contents

- Introduction
- Issues in the Design of a Code Generator
 - Instruction Selection
 - Register Allocation and Assignment
 - Instruction ordering
- A Simple Target Machine Model
- A Simple Code Generator : Code Generation for Trees
- Addresses in the Target Code
- Basic Blocks and Flow Graphs
- Optimization of Basic Blocks
- Liveness Analysis

Introduction



- **Requirements**

- Preserve semantic meaning of source program
- Make effective use of available resources of target machine
- Code generator itself must run efficiently

- **Challenges**

- Problem of generating optimal target program is undecidable
- Many subproblems encountered in code generation are computationally intractable

Issues in designing code generators

- Input to code generation
- Target program(output)
- Memory management
- Instruction selection
- Register allocation
- Evaluation order

Issues in the Design of a Code Generator

- **Inputs** – are IRs or simply intermediate code in the forms of:-
 - Graphical presentation (**syntax trees**, **DAGs**,...)
 - Linear presentation (**postfix**, ...)
 - three-address presentations (**quadruples**, **triples**, ...)
 - Virtual machine presentations (**bytecode**, **stack-machine**, ...)
- **Outputs** - depending on the requirements
 - **Absolute machine code** (executable code by the loader)
 - **Relocatable machine code** (object files for linker)
 - **Assembly language** (facilitates debugging)
 - **Byte code forms for interpreters** (e.g. JVM)

Issues in the Design of a Code Generator...

- **Three primary tasks**
 - *Instruction selection*
 - choose appropriate target-machine instructions to implement the IR statements
 - *Register allocation and assignment*
 - decide what values to keep in which registers
 - *Instruction ordering*
 - decide in what order to schedule the execution of instructions
- **Design of all code generators involve the above three tasks**
- **Details of code generation are dependent on the specifics of IR, target language, and run-time system**

Instruction selection

- The complexity of mapping IR program into code-sequence for target machine depends on:
 - *Level of IR (high-level or low-level)*
 - *If IR is high-level* - each IR statement is translated into a sequence of machine instructions. Poor code produced – requires optimization
 - *If IR is low level* - low-level details of the underlying machine can be used to generate more efficient code sequences
 - *Nature of instruction set (data type support)*
 - E.g. the uniformity and completeness of the instruction set are important factors
 - *Desired quality of generated code (speed and size)*
 - If efficiency of the target program were not an issue, instruction selection is straightforward

Example

- $x = y + z$ this is three code address representation

Mov y,R0

ADD z,R0

Mov R0,x

- $a = a + 1$

Mov a,R0

ADD#1,R0

Mov R0,a

- $a = b + c$

- $d = a + c$

Mov b,R0 ADD c,R0 Mov R0,a

Mov a,R0 ADD c, R0 Mov R0,d

From the above two moving instruction are similar so it will take the first one only.

Mov b,R0 Mov c,R0 ADD c,R0 Mov R0,d

This one is more efficient relative to the first one

Instruction selection ...

- Given IR it can be implemented by many different code sequences, with significant cost differences between the different implementations.
 - Example: let's take statement $a = a + 1$
 - It can be implemented in either of the following ways
 - INC a* // if the target machine has an "increment" instruction (INC)
 - OR
 - LOAD RO, a* // $RO = a$
 - ADD RO, RO, #1* // $RO = RO + 1$
 - STORE a, RO* // $a = RO$
- Which code is efficient?
- Of course, We need to know instruction costs in order to design good code sequences

Register allocation

- We apply it when we want to perform more number of operation and we have limited numbers of register.
- So how can we allocate with limited number of register more number of operation?

It performed by two ways

- ✓ Register allocation:- it specify which register contains which variable let registers R0, R1 R0 contain which variable and R1 contain which variable
- ✓ Register assignment:- it is the opposite of register allocation which means which variable contain which register like a contains R0 and b contains R1

Register Allocation

- A key problem in code generation is deciding what values to hold in what registers
- Registers are the fastest computational unit, but we usually do not have enough of them to hold all values
- The use of registers is often subdivided into two sub problems:
 - **Register allocation**, during which we select the set of variables that will reside in registers at each point in the program.
 - **Register assignment**, during which we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single-register machines.
 - The problem is NP-complete
 - Have to follow register usage of hardware & Operating Systems

Register Allocation...

- What variables can the allocator try to put in registers?
 - **Temporary variables: easy to allocate**
 - defined & used exactly once, during expression evaluation implies allocator can free up register when done
 - usually not too many in use at one time implies less likely to run out of registers
 - **Local variables: hard, but doable**
 - need to determine last use of variable in order to free register
 - can easily run out of registers implies need to make decision about which variables get register allocation
 - **Global variables**
 - really hard, but doable as a research project

Cont...

Example

$t = a + b$

$t = t * c$

$t = t / d$

Mov a,R0

ADD b,R0

Mul c,R0

Div d,R0

Mov R0,t

Mov a,R0

ADD b,R0

Mov R0,t

Mul c,R0

Div d,R0

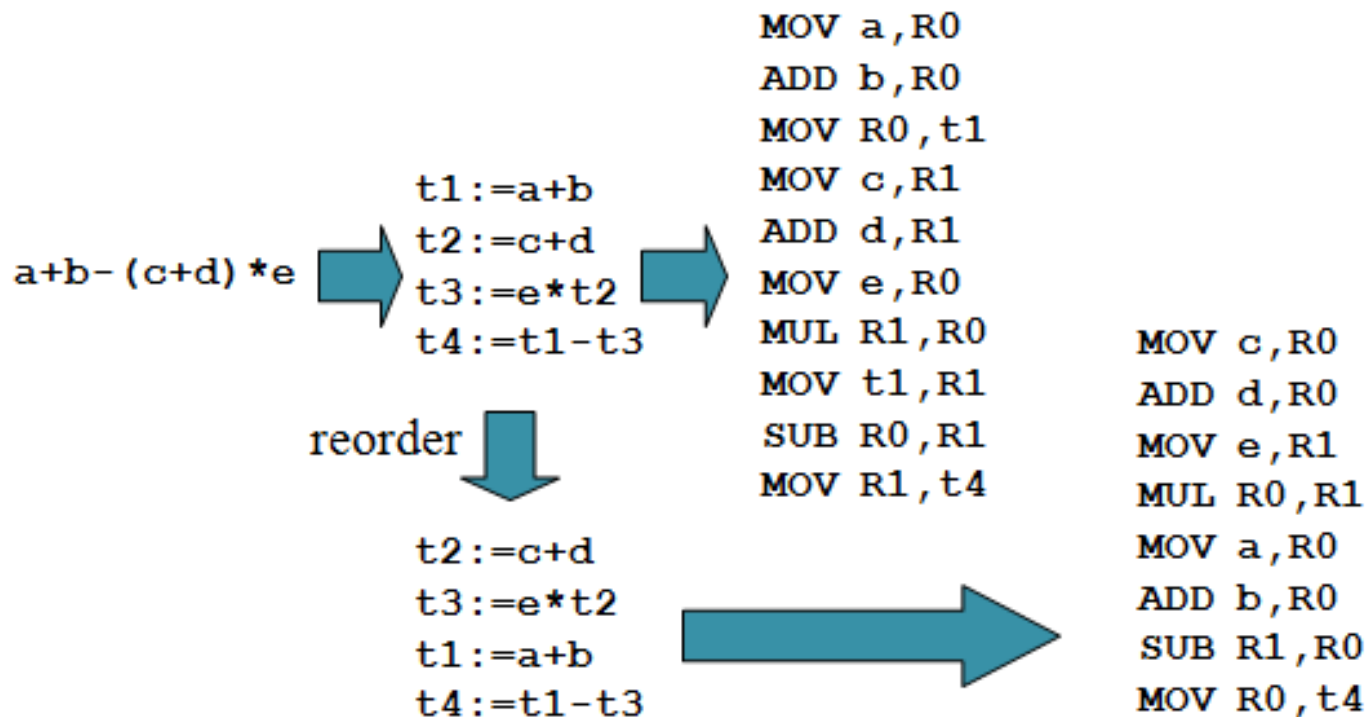
Mov R0,t

...

Select the efficient one

Evaluation Order

- Is selecting the order in which computations are performed
- Affects the efficiency of the target code - some computation orders require fewer registers to hold intermediate results than others.
- Picking a best order is NP-complete
- **Example**




A Simple Target Machine Model

- Implementing code generator requires through understanding of the *target machine architecture* and its *instruction set*
- Our (hypothetical) machine:
 - Byte-addressable (word = 4 bytes)
 - Has n general purpose registers $\mathbf{R}_0, \mathbf{R}_1, \dots, \mathbf{R}_{n-1}$
 - Assume all operands are integers
 - Two-address(load , store) instructions of the form
op source, destination
 - Where **Op** is operation code and **Source, destination** are data fields
 - Computation operations: **OP dst, src1, src2**
 - Where **OP** is a operator like ADD or SUB, and **dst, src1** , and **src2** are locations, not necessarily distinct.


A Simple Target Machine Model

- Unconditional jumps: **BR** L where L is label
- Conditional jumps of the form **Bcond** r, L
 - where r is a register, L is a label
- A variety of addressing modes
 - variable name
 - $a(r)$ means $\text{contents}(a + \text{contents}(r))$
 - $*a(r)$ means: $\text{contents}(\text{contents}(a + \text{contents}(r)))$
 - immediate: $\# \text{constant}$ (e.g. LD R1, #100)
- Cost
 - cost of an instruction = $1 + \text{cost of operands}$
 - cost of register operand = 0
 - cost involving memory and constants = 1
 - cost of a program = sum of instruction costs

A Simple Target Machine Model...

$X = Y - Z$ 

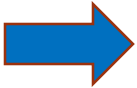
| | | |
|-----|------------|-----------------|
| LD | R1, y | // R1 = y |
| LD | R2, z | // R2 = z |
| SUB | R1, R1, R2 | // R1 = R1 - R2 |
| ST | x, R1 | // x = R1 |

$b = a[i]$
(8-byte elements) 

| | | |
|-----|-----------|------------------------------------|
| LD | R1, i | // R1 = i |
| MUL | R1, R1, 8 | // R1 = R1 * 8 |
| LD | R2, a(R1) | // R2 = contents(a + contents(R1)) |
| ST | b, R2 | // b = R2 |

$x = *p$ 

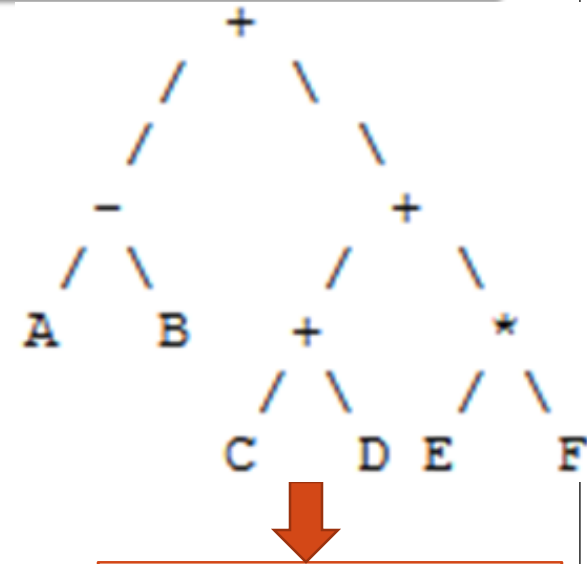
| | | |
|----|-----------|------------------------------------|
| LD | R1, p | // R1 = p |
| LD | R2, 0(R1) | // R2 = contents(0 + contents(R1)) |
| ST | x, R2 | // x = R2 |

if $x < y$ goto L 

| | | |
|------|------------|------------------------|
| LD | R1, x | // R1 = x |
| LD | R2, y | // R2 = y |
| SUB | R1, R1, R2 | // R1 = R1 - R2 |
| BLTZ | R1, M | // if R1 < 0 jump to M |

A Simple Code Generator : Code Generation for Trees

- Generate an assembly code, for the expression $(A-B)+((C+D)+(E*F))$
- The expression corresponds to the following AST and assembly code is
- We used only two registers
- There is an algorithm that generates code with the least number of registers. It is called the **Sethi-Ullman** algorithm.
 - It consists of two phases: **numbering** and **code generation**.



```
LD R1, C
ADD R1, D
LD R0, E
MULT R0, F
ADD R1, R0
LOAD R0, A
SUB R0, B
ADD R0, R1
```

Simple code generator

- It Generate target code for the sequence of three address statements.
- It used *getReg function* to assign registers to a variables
- For each operator in the statement there is corresponding target code operators.
- The two data structure of Simple code generator
 - ✓ **Register and address descriptors**
 - *Register descriptor*
 - ✓ It used to keeps the track of which variable is stored in a register . Initially all registers are empty.
 - *Address descriptor*
 - ✓ It used to keeps track of location where variable is stored. Location may be register a stack location or memory address.

Code generation algorithm

- For the given three address code $x = y \text{ op } z$
 - Invoke the function `getReg` to determine the location L where result of $y \text{ op } z$ store. Where L may be register/memory
 - Consult the address descriptor for y to determine y' , the current location of y . If y is not already in L , generate `MOV y' , L` .
 - Generate the instruction `op z' , L` update address descriptor of x to indicate that x is in L . if L is register update its descriptor to indicate that it contains the value of x
 - If y and z have no next use and not live on exit update the descriptor to remove y & z .

Example

- $d = (a-b) + (a-c) + (a-c)$
- The three address code representation is

$$t1 = a - b \quad t2 = a - c \quad t3 = t1 + t2 \quad d = t3 + t2$$

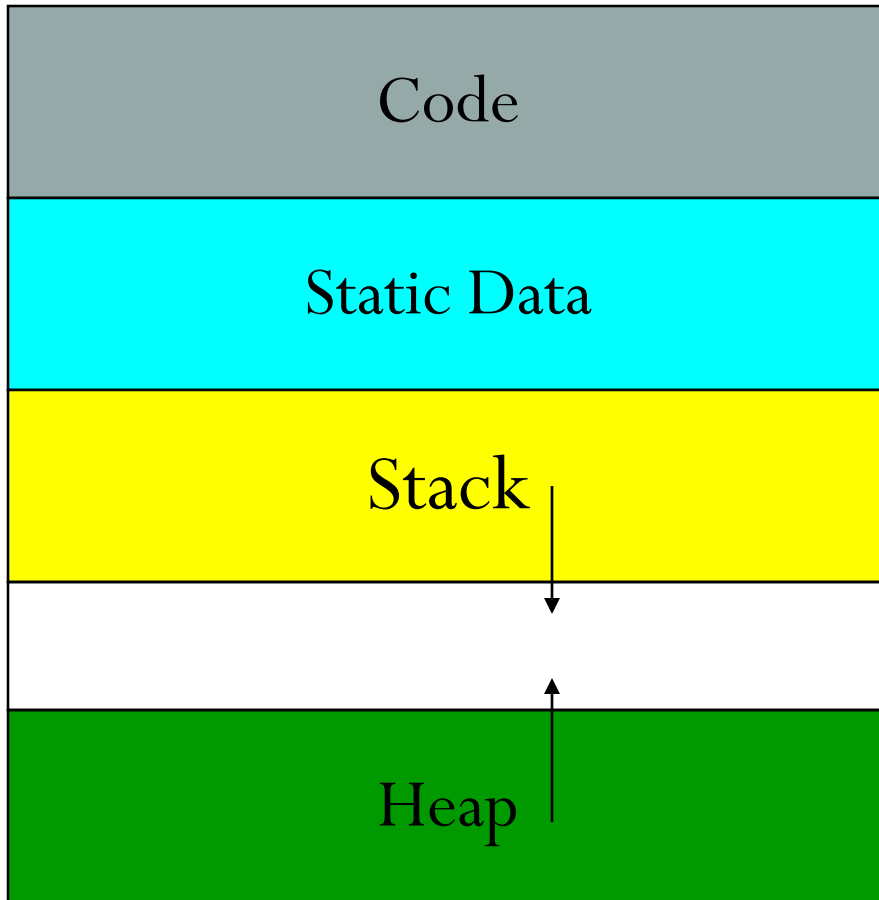
Only two registers are used

| Statement | code Generated | Register descriptor | Address descriptor |
|----------------|-----------------------|-------------------------------------|-------------------------------|
| $t1 = a - b$ | Mov a,R0 Sub b,R0 | Register are empty R0 contain t1 | t1 in R0 |
| $t2 = a - c$ | Mov a,R1 Sub c,R1 | R0 contain t1 R1 contain t2 | t1 in R0 t2 in R1 |
| $t3 = t1 + t2$ | ADD R1,R0 | R0 contain t3 R1 contain t2 | t2 in R1 t3 in R0 |
| $d = t3 + t2$ | ADD R1,R0 Mov R0,d | R0 contain d | d in R0 d in R0 and memory |

Addresses in the Target Code

- 4 distinct Regions of Memory
 - **Code space** – Instructions to be executed
 - Best if read-only
 - **Static (or Global)**
 - Variables that retain their value over the lifetime of the program
 - **Stack**
 - Variables that is only as long as the block within which they are defined (local)
 - **Heap**
 - Variables that are defined by calls to the system storage allocator (malloc, new)

Addresses in the Target Code...



- Code and static data sizes determined by the compiler
- Stack and heap sizes vary at run-time
 - Stack grows downward
 - Heap grows upward
- Some machines have stack/heap switched

Addresses in the Target Code...

- **Standard (simple) approach**

- Globals/statics – memory
- Locals
 - Composite types (structs, arrays, etc.) – memory
 - Scalars
 - Most frequently used – register
 - Rest – memory

- **All memory approach**

- Put all variables into memory
- Compiler register allocation relocates some memory variables to registers later

Thank you
Your final will [3-7]